Comp 167: Introduction to Computational Biology — Donna Slonim
Tufts University — Spring 2022

Homework 3

Exercises:

These are ungraded, short questions that are representative of the types of problems you might expect to see on quizzes. You do not need to submit written answers to these, but be prepared to discuss them in class.

1. What is the major limitation of sequencing technologies that makes it necessary to use computational methods to assemble genomic sequence?
2. What is the difference between overlap graphs and de Bruijn graphs? What are the underlying computational problems associated with each? What are advantages and disadvantages of each for use in sequence assembly? What happens to each in the presence of repeated sequence?
3. For what purpose would suffix trees likely be used in a sequencing project with long reads and relatively low coverage?
4. What is the key property of a database that allows methods like compressive BLAST to perform searches in expected sublinear time?
5. Suppose you are using the star alignment MSA algorithm. Once you have chosen your center sequence, does the order in which you merge the sequences into the alignment matter? Why, or why not?

Part I:

1. Suppose you are trying to assemble a sequence with the following sequence reads:

```
R1: CAGTTACCA
R2: TATACCAG
R3: CATCATTG
R4: CCAGGCAGT
R5: ACCATCA
```

You may assume that there are no errors in the sequence reads, that all the reads come from the same strand, and that every letter in the underlying sequence is represented somewhere in at least one of the reads.

1.1 (4 points) Draw the de Bruijn graph for this set of sequences, with k=4.

1.2 (8 points) Are there any repeated substrings of length 4 in the sequence? Of length 5? How does the graph support your answer to these questions? Is there an Eulerian path?

1.3 (4 points) Show the probable assembled sequence. Draw the location of each of the reads below the assembled sequence.

1.4 (4 points) How many branching nodes are there in the graph? (For purposes of this assignment, we will consider a branching node to be any node with indegree or outdegree greater than 1.) Mark them.

1.5 (6 points) Now let's use the graph, which we'll call $\mathcal{G}$, to identify non-ambiguous sequence contigs. Imagine removing all the branching nodes and their adjacent edges from $\mathcal{G}$ to create a new graph, $\mathcal{G}'$.

Start at any node with indegree 0 in $\mathcal{G}'$. Walk along the graph to construct a string of assembled sequence, starting with the k-mer at the node with indegree 0, and appending the last character at each node visited until you reach either a node you have visited before on this walk, or a node with outdegree 0. This string should be considered to be one "contig." Now imagine removing all the nodes and edges visited so far from $\mathcal{G}'$ and repeating the above procedure to find additional contigs.

List all the contigs you find in this graph.

1.6 (4 points) What percentage of the assembled sequence (question c) is "covered" by these contigs?

1.7 (6 points) Draw the overlap graph for this set of sequence reads. For the sake of readability, do not show zero-weight edges.

1.8 (6 points) Use a greedy algorithm to find a Hamiltonian path through this graph. To what assembled sequence does this path correspond?

2. Suppose, instead, that R4 is just slightly shorter:

```
R1: CAGTTACCA
R2: TATACCAG
R3: CATCATTG
R4: CCAGGCA
R5: ACCATCA
```

2.1 (4 points) How would this change affect your overlap graph and the corresponding greedy assembly?

2.2 (4 points) How would this change affect your de Bruijn graph and the corresponding assembly?

2.3 (4 points) Show the sequence contigs from this de Bruijn graph, following the approach from problem 1e.

2.4 (4 points) What is the N50 size of this set of contigs? How does it differ from the N50 size of the contigs you found in question 1e?

3. Suffix trees.

3.1 (4 points) Draw the suffix tree for the string RATATAT.

3.2 (8 points) Describe an algorithm for using suffix trees to find the longest repeated substring in a sequence. Hint: look at the suffix tree of RATATAT to get some ideas. Then generalize your intuition to design an algorithm.

Part 2: Sequence assembly.

For this part of the assignment, you will do simple error detection and partial assembly of a set of DNA sequence reads. You will need to implement a de Bruijn graph, as well as an error detection algorithm and an assembly algorithm, both of which will operate on your graph implementation.

Your initial value of k for the de Bruijn graph will be 31, although you should design your code to allow you to change this value.

You will first filter the input reads to determine whether any read contains a k-mer that occurs only once in the entire input. We suggest using the de Bruijn graph architecture to do this, given that you need to build the graph anyway.

The aim of this stage is to pproduce (and submit) a file called `good_reads` that contains only reads all of whose k-mers appear at least twice in the input. (A k-mer that occurs only once has a high chance of containing a sequencing error.)

Next, you will build a de Bruijn graph by adding each of these good reads into the graph. You will then identify any branching nodes. Remove the branching nodes and their associated edges from the graph, leaving a graph that will have multiple connected components. Mark all the visited nodes as read, and continue with the next source that has not yet been marked.

Implementation details appear below, but this should be enough to get many students started. For more information, read on...

Technical Details about your implementation:

You will find the reads you will be working with in the file
`/comp/167/public_html/private/hwkfiles/hwk3/sequence_reads`;
which is also downloadable from the course materials web page.

This file is simply a newline- delimited ASCII file, containing capital letters to represent each of the four nucleotides. The reads we are working with contain sequencing errors (an A instead of a T, for example) at a low rate, and the reads are an average of 100bp long. Repetitive sequences will result in ambiguity that your assembler will not be able to resolve: you will see these sections in your graph as branches and cycles.

To slightly simplify what you have to code, we will give you a tool you may use to create the graph information compiled from all the reads in an input file.

Specifically, in the same directory as the sequence_reads file, there is a compiled program named Builder_Helper that will run on homework.cs.tufts.edu. This program will create an output file file named graph.txt in the same directory where you run it. The file graph.txt will contain 3 lines for each vertex of the graph: the full k-mer string represented at that node, the k-mer(s) at each incoming node, and the k-mer(s) at each outgoing node.

The output from Builder_Helper will provide the structure of the De Bruijn graph. However, you will still need the text file of the sequence reads to count how often each k-mer occurs. Having the graph structure should simplify parsing the reads, allowing you to just put the graph together, find and remove branching nodes, and then traverse each component to find reads.

Builder_Helper takes its inputs on the command line, with no flags: the first argument is the name of a text file containing sequence reads, and the second is the intended size k for the graph. For example,

```
Builder_Helper good_reads.txt 31
```

De Bruijn graph details:

You will still need to be able to build a de Bruijn graph from the information in the graph.txt file.

For the Bruijn graph, please start with $k = 31$ for the length of the string at each vertex, but make sure your code is flexible; to answer the questions you will need to look at the results with different k. (You can run Builder_Helper with different k in that case.)

Here are some suggestions about the implementation. Each node in the graph must have an associated string (a k-mer) that the node represents, and it should maintain lists of both outgoing and incoming edges. For the algorithms you will be implementing, you may find it helpful to include two Booleans, one that can serve as a marker at that node, and one that indicates if the node is a branching node (details below). You may also want to store an integer to count the number of times that the node's string appears in the reads you are processing - if so, you can use this to filter the reads in your initial reads file to create the good_reads file.

You will need a way to iterate over all the nodes in the graph; one option is to store them in some sort of table, keyed by their strings.

If you want to use the graph to filter the reads, build an initial de Bruijn graph using the data from all the original sequence reads. Then go through each input read to identify any whose k-mer nodes appear only once. Output the remaining "good" reads (all of whose k-mers appear multiple times) to a file called `good_reads`. (You can however filter the reads another way, if that is easier for you.)

Unambiguous sequence assembly details:

After removing the erroneous reads, you will rebuild the de Bruijn graph, this time using only the reads in good_reads as input. You can use Builder_Helper to create a new graph structure, and then write code that lets you use this graph to assemble the reads into contigs of length at least 100bp.

Recall that repetitive genomic regions result in cyclic regions in the de Bruijn graph. Advanced assemblers like SOAPdenovo use mate-pair information and scaffolds to resolve these regions, but we are not doing that here. For this assignment, all you need do is output "contigs" corresponding to unambiguous regions of the de Bruijn graph.

To do this, first ensure that it is easy to tell when a node is a "branching" node. For the purposes of this assignment, a branching node is one with indegree or outdegree greater than 1. You may mark the nodes as branching nodes during graph construction or iterate through the graph nodes to identify them at this point.

Now, starting with a source node, walk through the graph, marking each node as you visit it, until you reach either a branching node or a node that has already been marked. This walk produces a linear path through the graph, which corresponds to a contig in the sequence assembly. (The string can and should be constructed using the k-mers stored at each vertex in the walk. Do *not* include the sequence from that branching node in your contig.)

Repeat this process until you have marked all the nodes in the graph from which branching nodes have been removed.

Using this algorithm, you will deterministically generate a set of unambiguous, contiguous genomic regions. As your output for this section of the assignment, you will produce two files. Write all such contigs longer than 100 bp, separated by newlines, to a file called `output_contigs`. The output format should be the same as the input read file, i.e., newline separated blocks of ASCII nucleotide text. Second, create a file called `contig_lengths`, containing the lengths of each of the contigs you wrote to the `output_contigs` file. This will make help you answer the questions below. Submit both files with your code.

How to Check your Work:

In order to verify the various components of your assembler are working, there is a small toy example, using more human-readable text rather than DNA sequence, in the directory

`/comp/167/public_html/private/hwkfiles/hwk3/demo/`;

You are given the original sequence for the demo (in originalseq.txt), the sequence_reads file, and the associated output files for that sequence (good_reads, output_contigs, contig_lengths). These files were generated using a contig minimum length of 8 and a k value of 5. You should be able to use Builder_Helper to generate the graph structure and reproduce the same output files.

What to Submit and How:

Submit your code in Gradescope. Make sure your code runs on the machine "homework" before you do so.

Please include a README file explaining how to compile (if necessary) and run your program.

Please also submit the three output files (`good_reads, output_contigs`, and `contig_lengths` described above.

Your code is worth 20 points, some of which will reflect its readability. (We are not grading you on algorithmic efficiency, but if your code is highly inefficient you may have trouble both testing it and answering the questions below.)

Please also answer the following questions. You may wish to put these answers in a separate text or pdf file and upload that file along with your code.

4.1 (20 points) Code, as described above.

4.2 (5 points) What is the N50 size for your assembly? (This is biased because you are not reporting contigs under 100bp, but that's okay.)

How much of your assembly is "chaff" (here, this can be defined as contigs of size under 100bp)?

4.3 (5 points) Try running your assembler with some different k values. What happens, and why do you think this is the case? Pay attention to the effects a change in k has on both error correction and on branching.